Maven

- Your project needs to be **independent**
  - Other developers need to be able to check out and run your project independently.
  - They shouldn't have to figure out, which JAR files to download, to run your project.
  - Maven helps with this problem.

- Maven is a build tool.
- What is a **build tool**?
  - The build tool is used to set up everything which is required to run your code independently.
  - It generates source code, compiling code, packaging code to a jar etc.

- **Maven** is a **build tool** used for building and managing **Java** Based Projects.

- **Maven** is based on the concept of a **POM (Project Object Model)**

- There are other Java Build Tools such as: **Ant, Gradle ...**

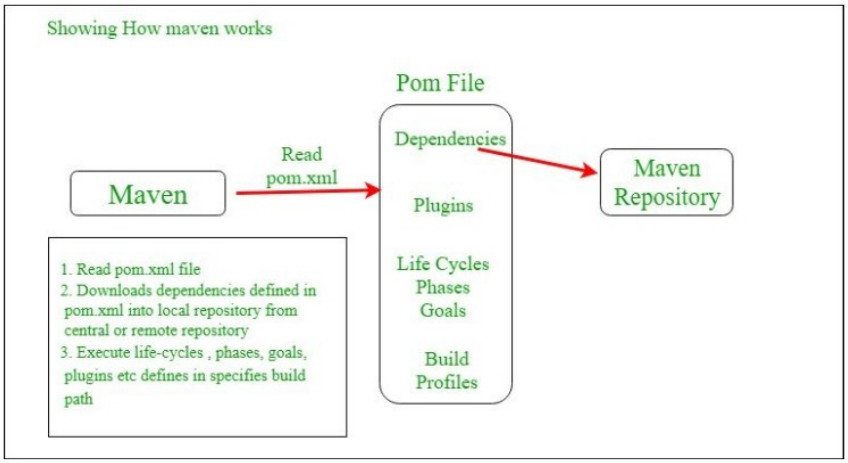
Maven with Selenium

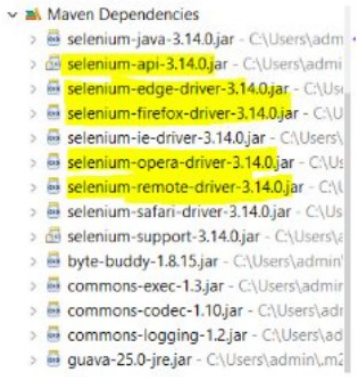
Automation testers used to face a common problem while using versions of JAR's/dependencies. All of their code was placed in Eclipse, IntelliJ and there was no central management of JAR versions etc.

Integrating Maven with Selenium solves that problem by providing following **benefits:**

- Maven is widely used for **dependency management** in Java.
  - **Dependencies** → are basically JAR files needed to run your project. *E.g: Selenium JAR is one dependency.*
  - Using **pom.xml** you can configure dependencies needed for building, testing and running code.
  - Maven **automatically downloads** the necessary files from the repository while building the project.
  - This way upgrading versions also becomes as easy as changing a number in the pom.xml file.
- It also provides a predefined folder structure to write the code.

How Maven Works





POM.XML

- Maven provides **pom.xml** which is the core of any project.
- It is an XML file that holds a list of the names of dependencies that our project uses.

```
<dependencies>
<!-- https://mvnrepository.com/artifact/org.seleniumhq.selenium/selenium-java -->
  <dependency>
      <groupId>org.seleniumhq.selenium</groupId>
      <artifactId>selenium-java</artifactId>
      <version>3.14.0</version>
  </dependency>
<!-- https://mvnrepository.com/artifact/org.testng/testng -->
```

- Dependencies are external Java libraries required for Project.
- Maven has an internal local repository, where it stores the Dependency JAR files. It's name is **M2**
  - Don't confuse it with your project GIT repository. It is just a folder in your computer, to store those JARs.
- If the dependencies are not found in the local Maven repository, Maven automatically downloads them from a central Maven repository and puts them in your local repository.

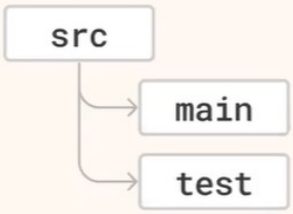
We put the code under test.

Pom.xml is under Maven project (Child).

We have to respect the conventinal structure.

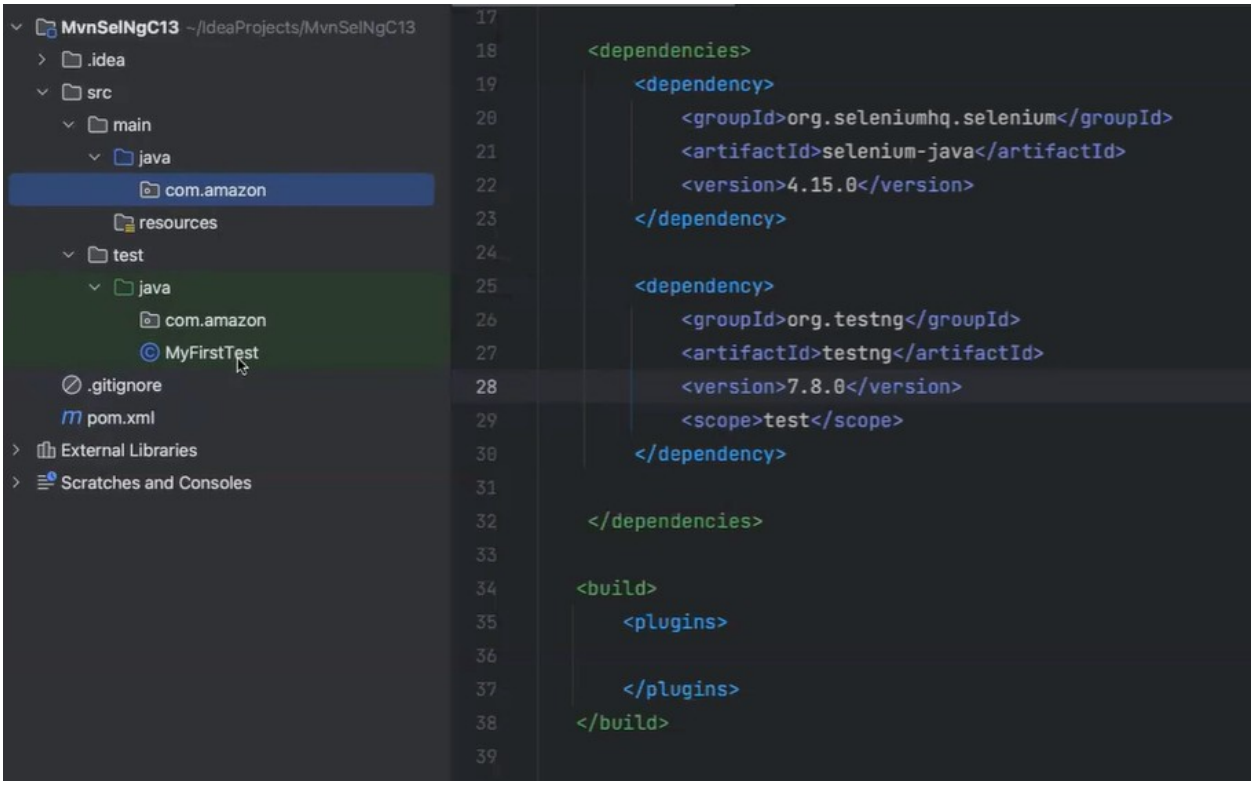
Project/Framework

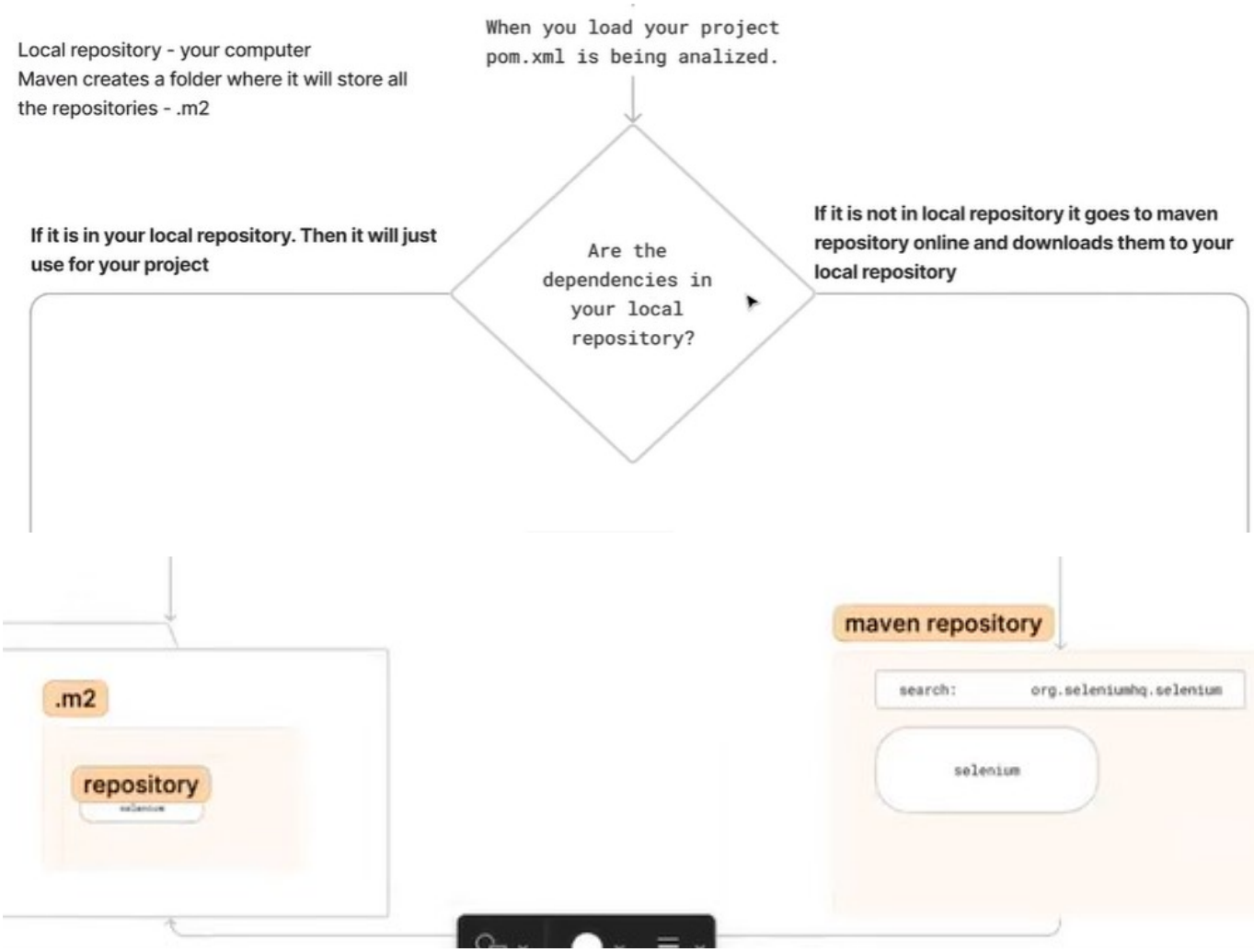
Folder Tree and Code Files



pom.xml

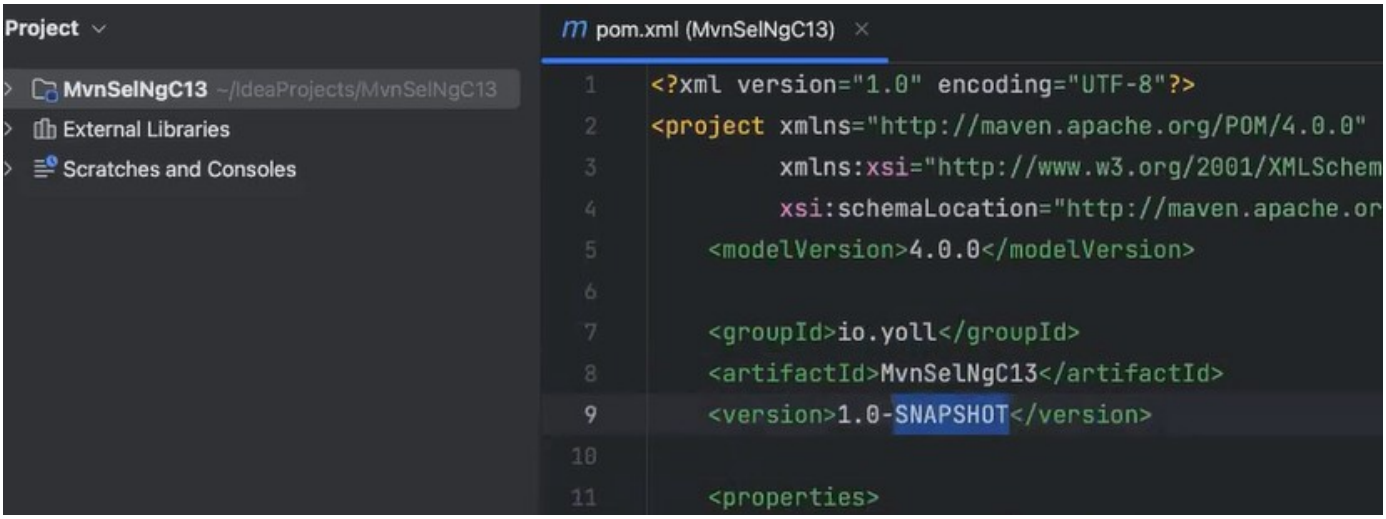
```
<dependency>
<groupId>org.seleniumhq.selenium</groupId>
<artifactId>selenium-java</artifactId>
<version>4.15.0</version>
</dependency>
```




Local repository - your computer
Maven creates a folder where it will store all the repositories - .m2
When you load your project pom.xml is being analized.
Are the dependencies in your local repository?
If it is in your local repository. Then it will just use for your project
If it is not in local repository it goes to maven repository online and downloads them to your local repository
.m2
repository
maven repository
search:
org.seleniumhq.selenium
selenium

General Phrases used in Maven:

- **groupId**: Generally groupId refers to domain id. For best practices **company name** is used as groupId.
    - E.g: io.Yoll -> amazon.com -> com.amazon
- **artifactId**: It is basically the name of the Jar without version. You can use project name here.
- **version**: This tag is used to create a version of the project.
- **Local repository**: Maven downloads all the required dependencies and stores in the local repository called m2.



What Does "SNAPSHOT" Mean?

- **SNAPSHOT** indicates a version of a project that is currently in development. It is not a final, stable release and can be updated or changed at any time.

Versioning in Maven

1. **Release Version**:
    - This is a stable version of your project that is considered final.
    - Example: 1.0.0, 2.3.1.
    - Once a release version is deployed to a repository, it should not be changed. Each release version is immutable.
2. **SNAPSHOT Version**:
    - This is a version under active development.
    - Example: 1.0.0-SNAPSHOT, 2.3.1-SNAPSHOT.
    - SNAPSHOT versions can be updated regularly and overwritten in the repository. They represent the latest work-in-progress for that version.

BUILD LIFECYCLE

Basic maven phases are used as below.

1. **validate:** check if all information necessary for the build is available
2. **compile**: used to compile the source code of the project.
3. **test**: test the compiled code and these tests do not require to be packaged or deployed.
4. **package**: package is used to convert your project into a jar or war etc.
5. **verify**: run any checks on results of integration tests to ensure quality criteria are met
6. **install**: install the package into the local repository for use of another project.
7. **deploy**: copy the package to the remote repository

Phases are executed in a specific order. This means that if we run a specific phase using the command:

mvn -test → **won't only execute the specified phase but all the preceding phases as well.**

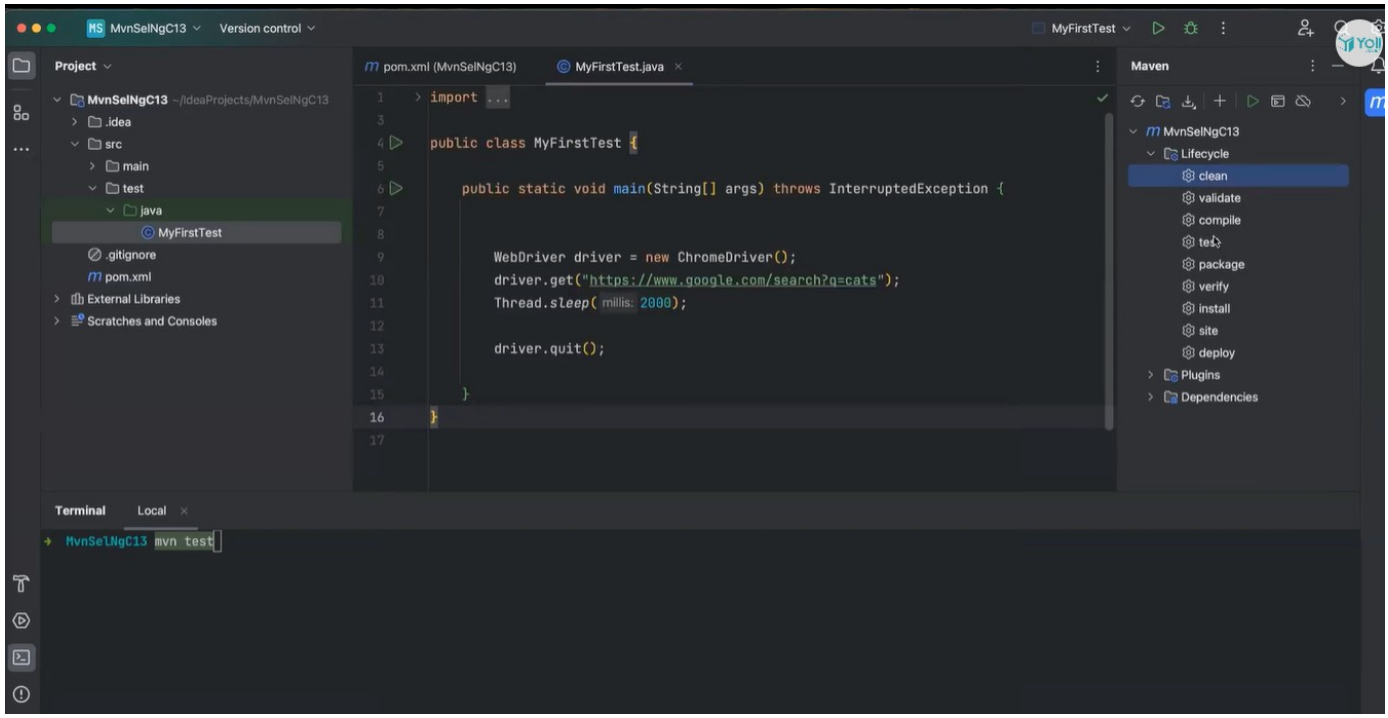
Click right on the corner to refresh what you put on pom.xml of dependency:

pom.xml (MvnSelNgC13)

```xml
<groupId>io.yoll</groupId>
<artifactId>MvnSelNgC13</artifactId>
<version>1.0-SNAPSHOT</version>

<properties>
    <maven.compiler.source>20</maven.compiler.source>
    <maven.compiler.target>20</maven.compiler.target>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>


<dependencies>
    <dependency>
        <groupId>org.seleniumhq.selenium</groupId>
        <artifactId>selenium-java</artifactId>
        <version>4.15.0</version>
    </dependency>

    <dependency>
        <groupId>org.testng</groupId>
        <artifactId>testng</artifactId>
        <version>7.8.0</version>
        <scope>test</scope>
    </dependency>

</dependencies>
```

Click on M to read what you have on:

- Lifecycle.
- Plugins.
- Dependencies.



The lifecycle of a project in IntelliJ IDEA, particularly when working with Maven projects, involves several phases that are executed in a specific order. Each phase represents a stage in the build process, and they are designed to manage the complete lifecycle of a project, from compiling the code to deploying the final artifact.

Key Phases in the Maven Build Lifecycle

1. **Clean**:
    - **Purpose**: Cleans the project by removing files generated in previous builds, such as compiled classes and jar files.
    - **Execution**: Typically, this phase deletes the target directory where Maven stores all generated files.
    - **Command**: mvn clean
2. **Validate**:
    - **Purpose**: Checks if the project structure is correct and all necessary information is available before proceeding with the build.
    - **Execution**: Ensures that the pom.xml is properly configured, and necessary files are present.

- **Command**: mvn validate

3. **Compile**:
   - **Purpose**: Compiles the source code of the project.
   - **Execution**: Takes the .java files and compiles them into .class files, placing them in the target/classes directory.
   - **Command**: mvn compile
4. **Test**:
   - **Purpose**: Runs the tests against the compiled code.
   - **Execution**: Uses frameworks like JUnit or TestNG to execute unit tests in the src/test/java directory.
   - **Command**: mvn test
5. **Package**:
   - **Purpose**: Packages the compiled code into a distributable format, such as a .jar or .war file.
   - **Execution**: After compiling and running tests, Maven packages the code, typically placing the artifact in the target directory.
   - **Command**: mvn package
6. **Install**:
   - **Purpose**: Installs the package into the local Maven repository, making it available for other projects on your machine.
   - **Execution**: The packaged .jar or .war file is copied to your local Maven repository (usually located in ~/.m2/repository).
   - **Command**: mvn install
7. **Deploy**:
   - **Purpose**: Deploys the packaged artifact to a remote repository, making it available for other developers or projects.
   - **Execution**: This phase is usually performed in a continuous integration/continuous deployment (CI/CD) pipeline.
   - **Command**: mvn deploy
8. **Site**:
   - **Purpose**: Generates project documentation and reports.

- **Execution**: Maven can produce a site with information like code coverage, dependency graphs, and project documentation, which can be viewed in a browser.
- **Command**: mvn site

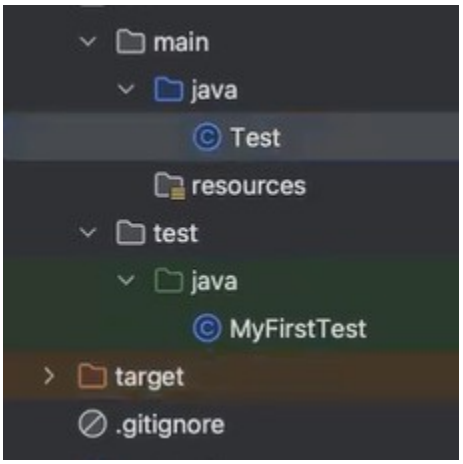
How These Phases Work Together in IntelliJ IDEA

In IntelliJ IDEA, these Maven lifecycle phases can be executed directly from the IDE, either by using the Maven tool window or by running Maven goals through the command line.

- **Clean Phase**: Typically executed before other phases to ensure a fresh start.
- **Validate, Compile, Test, Package, Install**: These are the core phases that are often executed sequentially during development. For instance, when you build a project (mvn install), Maven will automatically run validate, compile, test, and package before installing the artifact.
- **Deploy and Site**: These phases are more specialized and are typically used in environments where you're preparing for release or generating documentation.

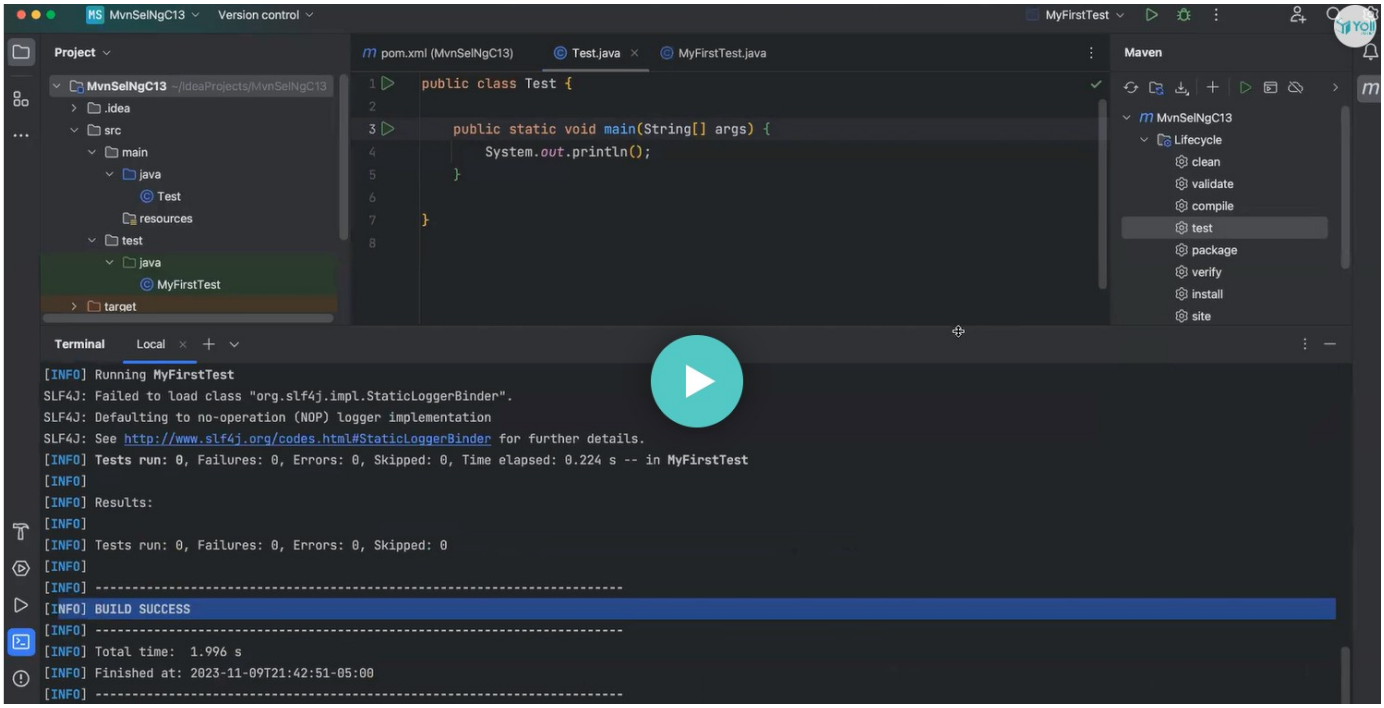
  TARGET is the FILE that evey command above execute with NOT THE JAVA CLASS

  EXAMPLE: WHEN USING CLEAN COMMAND, IT DOES NOT EFFECT THE JAVA CLASS PROGRAMMED.



Exemple

by clicking on TEST, or tape on Command Terminal: mvn test



We as SDET usually will use a portuion of maven cycles: mvn clean

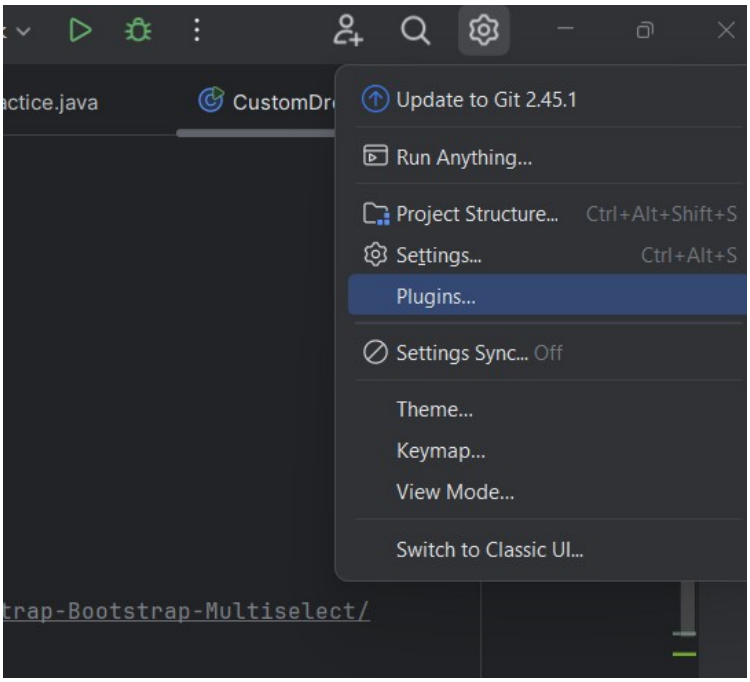
mvn clean install

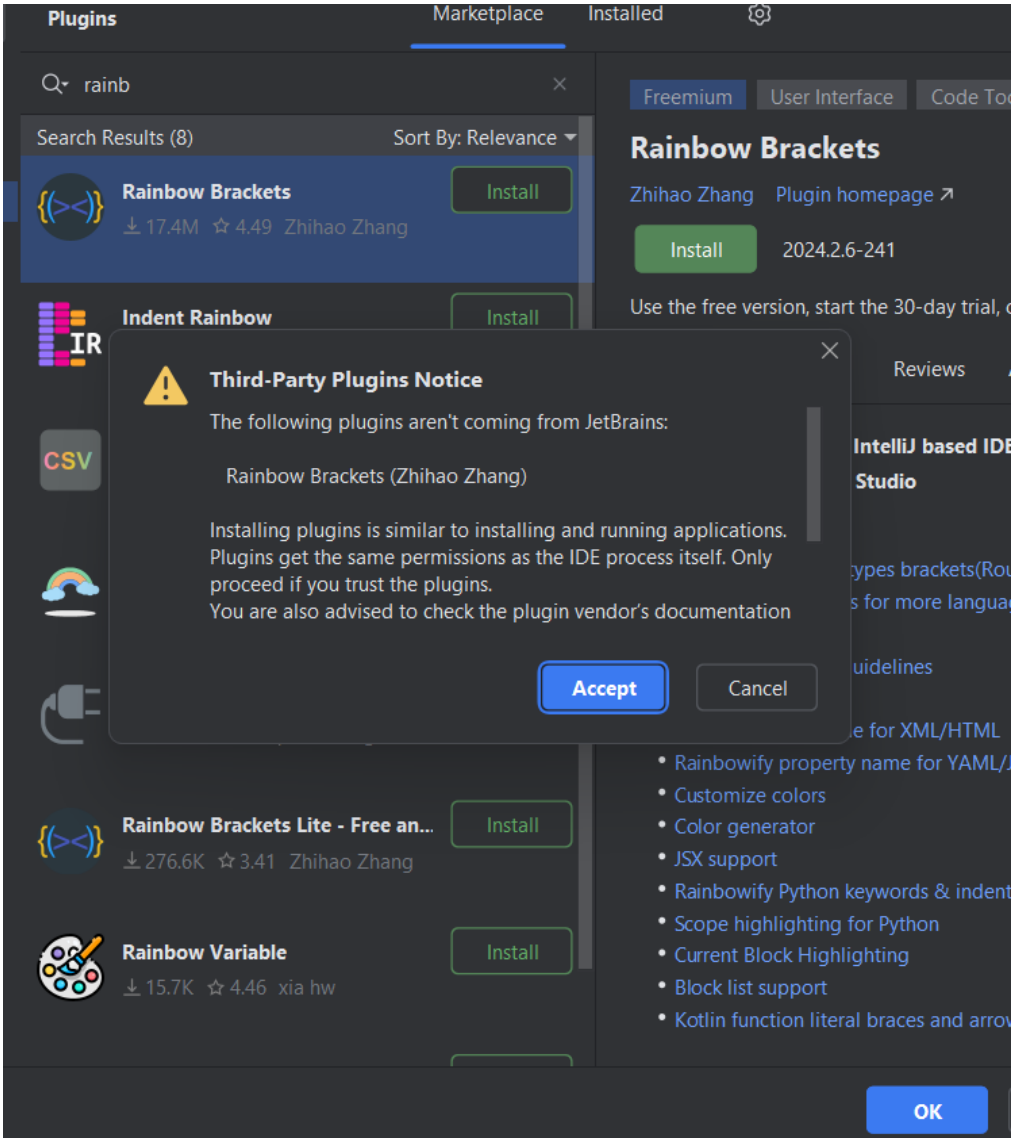
mvn clean test

mvn test

TO MODIFY THE UI OF INTELLIJ: LIKE THE BRACKET /Maven Dependency Checker /Atom

GO TO Settings, Click on plugins/MarketPlace/ Choose Rainbow Brackets





We as SDET usually will use a portuion of maven cycles:

mvn clean

mvn clean install

mvn clean test

mvn test

▸ How maven works?

▾ Questions:

▾ What is a - Build tool, and why we need it?

We use build tools to automate tasks such as compiling code, packaging, running tests, and creating documentation.
Doing these tasks manually would take a lot of time and could lead to mistakes.
Build tools ensure that the software is compiled and built consistently, making the development process more efficient and reliable.

▾ What Build tools we know for Java projects?

- Maven
- Ant
- Gradle

▾ What is maven?

Maven is a program that helps developers build and manage Java-based software.

It automates the process of putting together the code and making sure all the parts of the software work well together.

The primary usage for SDET's will be:

- dependency/plugins management
- use of some cycles - clean, install, test
- use these commands for automation(explain Jenkins)
  - Smoke Suite - I want to run the suite each and every morning at 8 AM
  - Jenkins - Cloud CI/CD tool - manages the delivery of the apps automatically
  - Go to may maven project - Take the code that is currently in GitHub then run the command "mvn clean test" - run it everyday at 8 AM EST

▾ What is a dependency in Maven context?

A dependency is like an additional tool or add-on (*a piece of external code*) that your project requires to complete a task.
For example, if your project involves running automated tests, you might use TestNG as your testing framework and Selenium to automate UI.

▾ What is the place that we manage all the dependencies and plugins in maven?

In Maven, you manage all the parts your project needs, like libraries and tools, in a file called `pom.xml`.

▾ How maven works on background, how it downloads dependencies?

1. **Reads the POM**: Maven starts by reading the `pom.xml` file of your project, where dependencies are listed.
2. **Checks Local Repository**: Maven first looks for dependencies in a local repository (.m2) on your computer, which is a cache of every dependency you've used before.
3. **Downloads from Central Repository**: If a dependency isn't found locally, Maven will download it from a central repository on the internet (mavenrepository.com), which is a large collection of commonly used libraries.
4. **Stores in Local Repository**: After downloading, Maven stores these dependencies in your local repository. Next time you need the same dependency, Maven won't have to download it again—it'll use the local copy.
5. *(not important) -* ***Handles Transitive Dependencies****: Dependencies often need other dependencies (called transitive dependencies). Maven automatically resolves these, downloading and storing them too.*

Maven provides a standard directory structure that is used across all Maven projects. By following this convention, Maven projects are consistent and easily understood by developers. Here's the standard layout:

```HTML
your-project
|-- pom.xml
|-- src
|   |-- main
|   |   |-- java
|   |   |   `-- (your package structure)
|   |   `-- resources
|   |       `-- (resources like properties files)
|   |
|   `-- test
|       |-- java
|       |   `-- (your test package structure)
|       `-- resources
|           `-- (test resources)
`-- target
    `-- (compiled code and packaged archives)
```

- `pom.xml` is the project's Project Object Model or configuration file for Maven.
- `src/main/java` is where you'd place your application's Java source code.
- `src/main/resources` contains files that are needed by your application but are not Java source code, like XML or property files.
- `src/test/java` is where the Java source code for tests is placed.
- `src/test/resources` holds resources needed for testing.
- `target` is where Maven places all the output from the build process.

▼ **What are lifecycles in maven?**

Lifecycles in Maven are like a to-do list for building and managing your software. They are made up of phases that do things like removing old files, creating new ones, checking code, and packing everything up. Maven has three main to-do lists (lifecycles) for different tasks: one for the main building process, one for cleaning up, and one for making documentation.

1. **Default Lifecycle**: Handles your project deployment, which includes compiling the code, packaging it into JAR or WAR files, running tests, and producing documentation.
2. **Clean Lifecycle**: Takes care of project cleaning, which removes all files generated by the previous build to ensure that the next build starts from a clean slate.
3. **Site Lifecycle**: Manages the creation of your project's documentation.

▼ **Explain more in details about default lifecycle.**

The default lifecycle in Maven is a series of steps to build your software, similar to the stages of constructing a house. Just as a house needs a solid foundation, sturdy walls, and a roof before it can become a home, Maven follows a set of steps to turn your code into a working application. Here's what each step does in the building process:

1. **validate**: Checks if all information necessary for the build is available and correct.
2. **compile**: Compiles the project's source code into binary code.
3. **test**: Runs tests using a suitable unit testing framework, such as JUnit or TestNG. These tests should not require the code be packaged or deployed.
4. **package**: Takes the compiled code and packages it in its distributable format, such as a JAR.
5. **verify**: Runs any checks to validate the package is valid and meets quality criteria.
6. **install**: Installs the package into the local repository, which can be used as a dependency in other projects locally.
7. **deploy**: Copies the final package to the remote repository for sharing with other developers and projects.

▼ **If we execute test phase with either terminal or maven control panel in your IntelliJ. How many phases will be executed overall?**

When you execute the `test` phase in Maven, all phases that precede `test` in the default lifecycle are also executed if they haven't been completed already in the current build.

These phases are executed in order:

1. **validate**
2. **compile**
3. **test**

▼ **Can we combine lifecycle phases in one command?**

Yes, you can combine Maven lifecycle phases in one command by listing them in the order they are supposed to be executed.

One of common command when you freshly set up you project/framework will be:

```HTML
mvn clean install
```

Another command that we usually will run:

```
mvn clean test
```